## 7.24

Do exercise

## 7.25

Do exercise

## 7.26

Arrays have 18 fill methods in the documentation. 2 of them take the int[] parameter

This is used to fill every element in the array with a value of your choice.

## 7.27

public Automaton(int numberOfCells)  
{  
 this.numberOfCells = numberOfCells**;** state = new int[numberOfCells + 1]**;** // Add an extra cell at the end  
 // Seed the automaton with a single 'on' cell in the middle. e.g. 00010000  
 state[numberOfCells / 2] = 1**;** state[1] = 1**;**}

The patterns of \* are the same but in a different position. At some point, the patterns collide, around 10 steps in.

## 7.28

right = (i + 1 < state.length) ? state[i+1] : 0**;**

left = (i == 0) ? 0 : state[ i-1 ]**;**

## 7.29

Created a new temporary nextState array because the values depend on the state array. But we would be modifying values in state array that we depend upon before we have had a chance to use the unmodified values.

We end up with only a single \* on each line if we only modify the existing state array. It does shift position though.

## 7.30 - Might have to come back to this as brain is not working this morning

You can use temporary variables to hold values until the position they are supposed to be inserted in of the old array is available, i.e. that element is no longer needed to determine another value.

I think you need to retain two temp variables, but my brain isn’t working this morning. The element in index-1 can be immediately replaced as it will no longer be needed. Then you hold 2 temp variables for centre and right.

Actually I think you only need to retain the centre variable as it is getting replaced. The right variable is not getting replaced.

Making a new array is clearer as you don’t risk modifying the starting array by accident. But it does use more memory and is possibly more computationally expensive. So it’s a trade off between clarity and efficiency.

## 7.31

Did that in the video already

public void update()  
 {  
 int left = 0**;** int center = state[0]**;** // start of the array  
 // Build the new state in a separate array.  
 int[] nextState = new int[state.length]**;** // Naively update the state of each cell  
 // based on the state of its two neighbors.  
 for(int i = 0**;** i < state.length -1**;** i++) {  
 // Check that you aren't at the end of the array, and init right to the next element, otherwise 0  
 //int right = (i + 1 < state.length) ? state[i+1] : 0;  
 int right = state[i+1]**;** // Modify the next element  
 nextState[i] = calculateNextState(left**,** center**,** right)**;** //nextState[i] = (left + center + right) % 2;  
 left = center**;** center = right**;** //int left, center, right;  
// if(i == 0) {  
// left = 0;  
// }  
// else {  
// left = state[i - 1];  
// }  
// center = state[i];  
// if(i + 1 < state.length) {  
// right = state[i + 1];  
// }  
// else {  
// right = 0;  
// }  
// nextState[i] = (left + center + right) % 2;  
 }  
 state = nextState**;** }

## 7.32

Did that in the video already

public int calculateNextState(int l**,** int c**,** int r) {  
 int[] next = new int[] {  
 0**,** 1**,** 1**,** 0**,** 1**,** 0**,** 0**,** 1 // From (l + c + r) % 2  
 }**;** int index = l \* 4 + c \* 2 + r**;** return next[index]**;**}

## 7.33

Some of those combinations wouldn’t have same result.

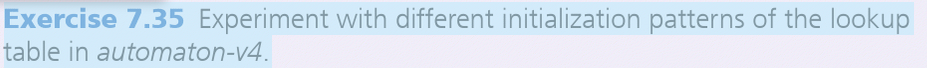
Did that in the video already

This is 3-bits so it has 8 possibilities (i.e. 2^3)

## 7.34

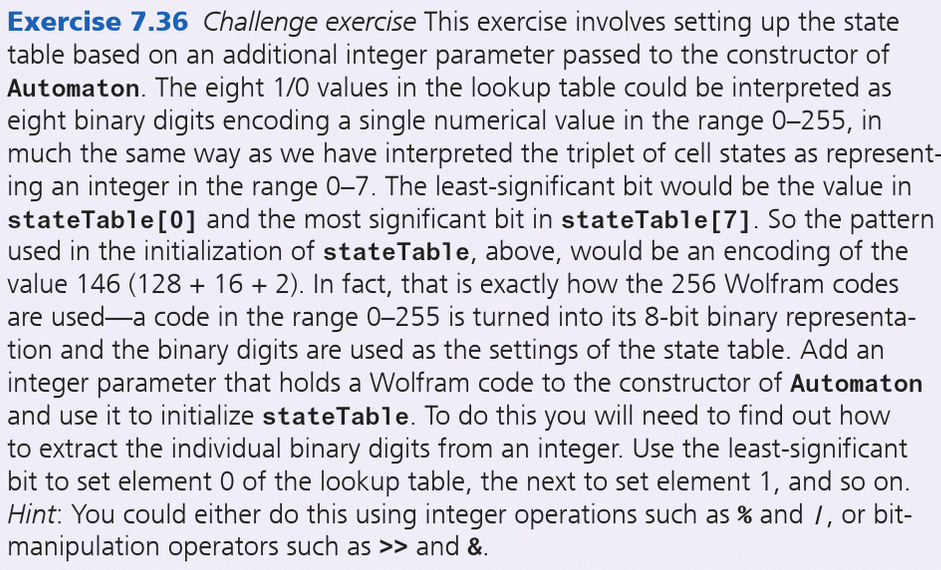
Did that in the video already see 7.31

## 7.35



public Automaton(int numberOfCells) {  
 this.numberOfCells = numberOfCells**;** // Allow an extra element to avoid 'fencepost' errors.  
 state = new int[numberOfCells + 1]**;** stateTable = new int[]{  
 // Original 0, 1, 0, 0, 1, 0, 0, 1  
 1**,** 1**,** 1**,** 1**,** 1**,** 1**,** 1**,** 1**,**  
 }**;** // Seed the automaton with a single 'on' cell.  
 state[numberOfCells / 2] = 1**;**}

## 7.36



This was a long way to ask, turn an 8-bit integer (0-255) into a series of binary code (but with the 128-place on the right instead of the left as normal)

E.g. 146 is { 0, 1, 0, 0, 1, 0, 0, 1 } starting with 2^0=1 on the left and ending with 2^7=128 on the right

int[] table = new int[8]**;**

if (wolfram >= 0 && wolfram < 256) {  
 int x = 128**;** for (int i = 7**;** i >= 0**;** i--) {  
 System.*out*.println("x: " + x + ", wolf: " + wolfram)**;** if (wolfram >= x) {  
 table[i] = 1**;** wolfram -= x**;** } else {  
 table[i] = 0**;** }  
 x /= 2**;** }  
} else {

// Do something if it is out of the range of 8-bit numbers

}

## 7.37

Do exercise

# 7.38 Game of Life

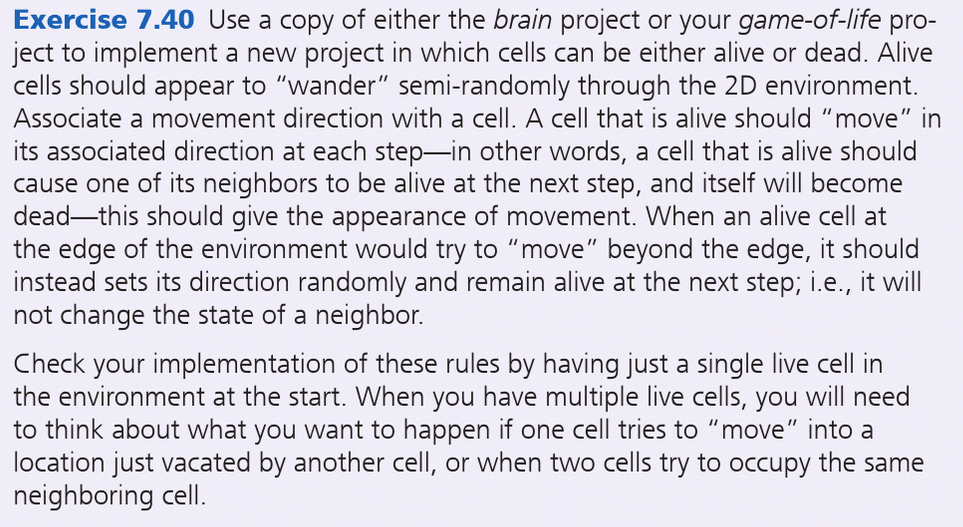
public int getNextState() {  
 int aliveCount = 0**;** for (Cell n : neighbors) {  
 if (n.getState() == *ALIVE*) {  
 aliveCount++**;** }  
 }  
  
 if (state == *DEAD*) {  
 return (aliveCount == 3) ? *ALIVE* : *DEAD***;** } else {  
 return (aliveCount < 2 || aliveCount > 3) ? *DEAD* : *ALIVE***;** }  
}

## 7.39

if (rowIndex + dr < 0 || rowIndex + dr >= numRows || colIndex + dc < 0 || colIndex + dc >= numCols) {  
 neighbors.add(new Cell())**; // Adds a dead cell here**} else {  
 neighbors.add(cells[rowIndex + dr][colIndex + dc])**;**}

The rules are still basically the same so it’s not so different when you look at individual cells. Overall though, the entire board dies a bit quicker because there are more dead cells overall.

## 7.40



This is longer than 4 hours

## 7.41 Different behaviours of “copying” an array including arraycopy

All except the assignment a = o has the original value

int[] o = new int[] { 11**,** 111**,** 1111 }**;**int[] a = o**;**int[] c = new int[3]**;**int[] clone = o.clone()**;**int[] arrayCopied = new int[3]**;**System.*arraycopy*(o**,** 0**,** arrayCopied**,** 0**,** o.length)**;**for (int i = 0**;** i < 3**;** i++) {  
 System.*out*.println("Original: " + o[i])**;** c[i] = o[i]**;** // assignment via iteration  
 o[i]++**;** // modify original  
 System.*out*.println("Original++: " + o[i])**;** System.*out*.println("Assignment copy: " + a[i])**;** System.*out*.println("ArrayCopy: " + arrayCopied[i])**;** System.*out*.println("Copied by iteration: " + c[i])**;** System.*out*.println("Cloned array: " + clone[i])**;**}

## 7.42

* asList will turn the array into a List type, which is an interface that ArrayList uses
* binarySearch will do a search for the value you want and return the index location it is located at. Array must be sorted for it to work
* fill means you make every element the same value that you specify
* sort the elements into order if it is a sortable type

## 7.43

int[] cat = new int[10]**;**Arrays.*fill*(cat**,** 5)**;**for (int x : cat) {  
 System.*out*.println(x)**;**}  
  
String[] pets = { "Cat"**,** "Kitten"**,** "Dog"**,** "Puppy"}**;**List<String> purr = Arrays.*asList*(pets)**;**System.*out*.println(purr)**;**int[] catLoaf = { 77**,** 33**,** 22**,** 100**,** 11**,** 0 }**;**Arrays.*sort*(catLoaf)**;**for (int kitty : catLoaf) {  
 System.*out*.println(kitty)**;**}  
int index = Arrays.*binarySearch*(catLoaf**,** 22)**;**System.*out*.println("index of 22: " + index)**;**

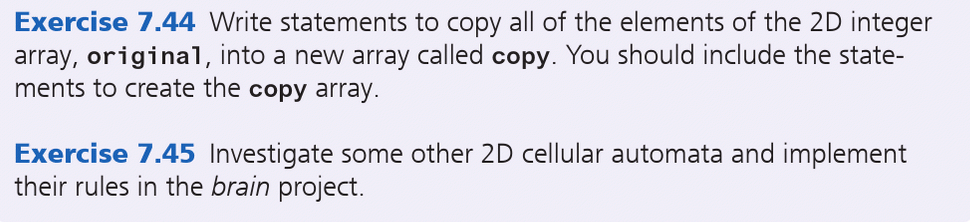
## 7.44

For fixed size arrays

int[][] original = { {0**,**0}**,** {1**,** 1}**,** {2**,** 2}**,** {3**,** 3} }**;**int[][] copy = new int[4][2]**;**for (int i = 0**;** i < original.length**;** i++) {  
 for (int j = 0**;** j < original[i].length**;** j++) {  
 copy[i][j] = original[i][j]**;** }  
}

For jagged arrays

int[][] copy2 = new int[original.length][]**;**for (int i = 0**;** i < original.length**;** i++) {  
 copy2[i] = new int[original[i].length]**;** for (int j = 0**;** j < original[i].length**;** j++) {  
 copy2[i][j] = original[i][j]**;** }  
}



## 7.45

What?